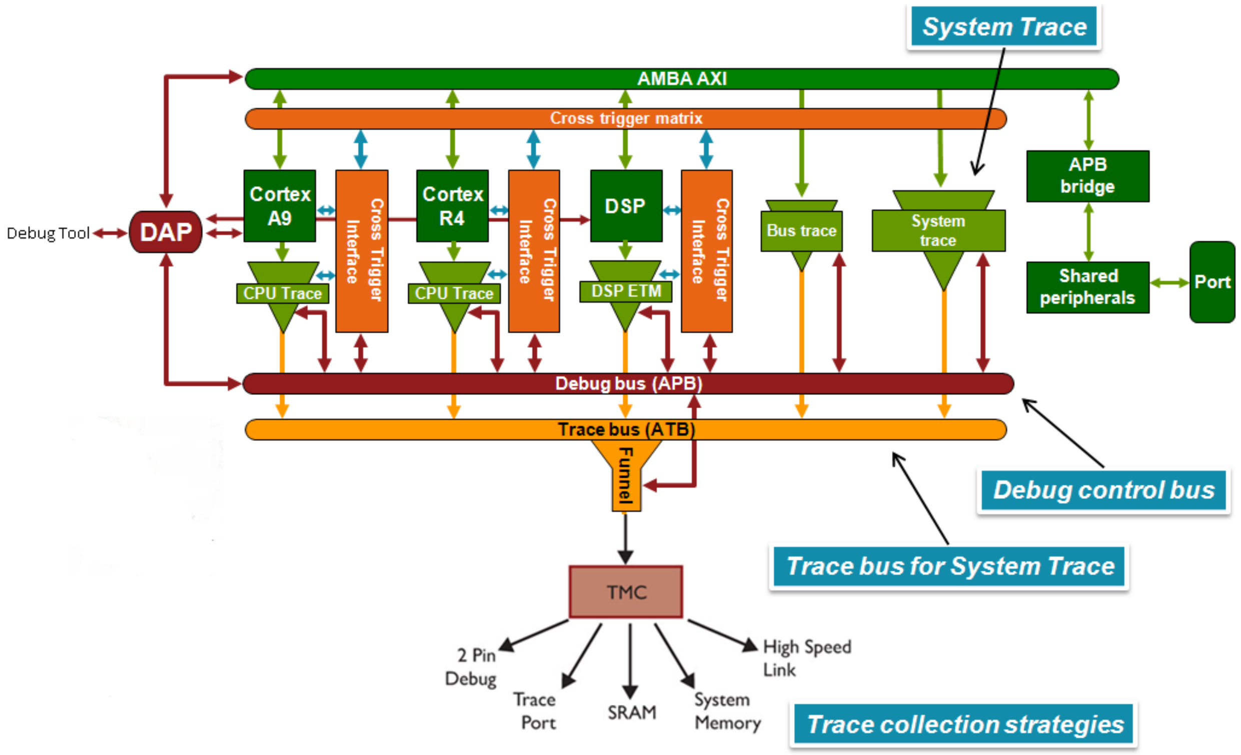
Debug and Trace using ARM® System Trace Macrocell (STM)



by Larry Traylor

SourcePoint™
More visibility. Software Debug and Trace.

By Larry Traylor, Vice President Software Debug and Trace

Larry Traylor co-founded Arium Corporation in 1977. Larry served as president, CEO, and chairman of the board of Arium. He was instrumental in driving that company's vision for product creation of hardware-based program debug and code trace tools. In 2013, Larry joined ASSET InterTech when Arium was acquired by ASSET. He has a BSEE from Cal-Poly Pomona.

Table of Contents

Figures and Tables

Executive Summary

Project delays caused by difficulties finding the cause of system-level failures are a major contributor to increasing the time-to-market for technical products. STM (System Trace Macrocell) can be a game changer by reducing the time it takes to find this type of problem.

Products containing software, which is almost every electronic product today, are becoming more complex along several dimensions, including multicore architectures, heterogeneous cores, multi-threading software, power management and other features as well as larger code bases. This complexity results in bugs occurring at system run time that are much harder to diagnose. To get these products to market in a timely fashion and in robust enough condition to create customer satisfaction, new debug features like STM must be supported.

Unfortunately, many of today' system-on-a-chip (SoC) devices contain application processors that do not produce data trace. (Data trace is a group of trace packets with the addresses and values of the data objects operated on by the program.) In addition, an SoC can often contain several cores and several different types of cores. Add these factors to hardware that changes power states (i.e. low power modes) as well as active clock domains and the result is that tracing the state of an SoC to find an error becomes very difficult.

The old model of simply tracing the executed instruction stream is far from sufficient to understand what is going on within the SoC. However, real-time tracing is still one of the most effective tools available for finding the root-causes of catastrophic symptoms.

For trace to be effective in complex SoCs, various types of events measured at various places within the SoC must be traced. STM is a newer trace element which, when integrated into an ARM® CoreSight® trace structure, can provide the added event and data value tracing necessary to render and observe changes in the state of the system.

The importance of firmware debug tools

Within the context of this document, debug refers to finding the root cause of an error that exhibits itself as a software error. The actual root-cause could occasionally be a hardware issue, but usually a software bug is also present. Trace is one of the most powerful debug tools.

The relative proportion of software development costs to hardware development costs is growing very fast. An ITRS (International Technology Roadmap for Semiconductors) report in 2011 presented the following data:

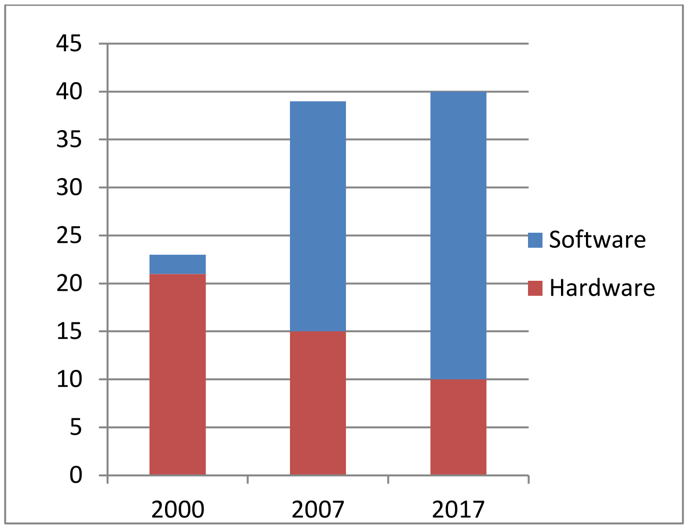


Figure 1: Relative Development Costs of a Product

This trend will likely continue over the next decade. The CoreSight architecture from ARM has been the world's leader in silicon tool hooks to allow firmware debug. STM is the next really big step.

What is an STM?

STM is a trace source (trace generator in an SoC) which provides an orthogonal means of tracking events and data that are not generated from processor execution trace sources. The STM that is considered in this document is the ARM version which is compliant to both CoreSight and the MIPI Alliance's MIPI STP-V2. (Other STMs are available, such as the TI-V1, but these are not addressed in this document.)

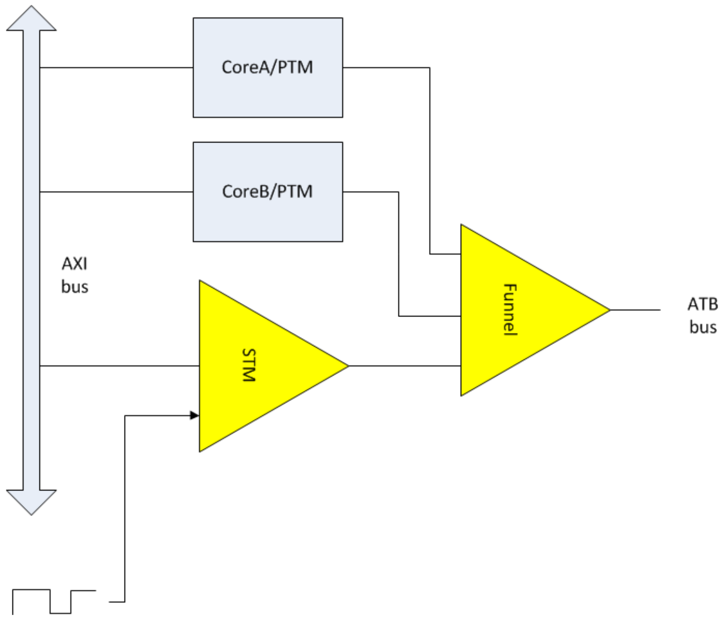


Figure 2: STM Architecture

The ARM STM has two primary types of interfaces which are used to stimulate the production of trace messages on the Advanced Trace Bus (ATB) (Figure 2). The Advanced eXtensible Interface (AXI) is a slave interface which, when written to, generates trace messages on the ATB bus. In addition, up to 32 hardware signals can be monitored and trace packets generated when the hardware signals change. Like most CoreSight components, STM is programmed (configured, started and stopped) via an Advanced Peripheral Bus (APB) interface. Starting and stopping STM can also be controlled by Cross Trigger Interface (CTI) connections. The output from STM is to an ATB bus. The following is a simple diagram of an STM application in a SoC.

What STM does for debugging engineers

STM is a trace element which allows for several types of trace messages to be added to a trace stream. This trace stream may or may not also contain instruction flow trace streams. Adding STM messaging has several effects. STM trace messages can be a way of tracing the high-level state of the SoC-based system. These trace messages can indicate what states the system transitioned through on its way to a point of interest (a symptom). Most bugs could be described as an incorrect or unexpected transition in these states. Historically, this type of function has been accomplished by "printf" statements, which were often directed to slow hardware ports or built-in logs. Logging this data through an STM is much faster as well as much more efficient in terms of the number of instructions needed to record the event.

STM messages can also contain the specific value of program objects. If a specific value that the program writes to a variable (data object) is needed, that value can be output via trace. This type of tracing is missing from instruction trace elements like the Program Trace Macrocell (PTM), which is often found on large SoCs containing application processors (A8, A9, etc.).

STM has an additional advantage of being core-type ambivalent. This means that system state messaging can come from completely dissimilar core/processor types, yet be time and order correlated in the same trace stream.

When STM tracing is added to instruction execution tracing, the trace stream can contain both the high-level system state indicators from STM as well as low-level detail from PTM and other sources to allow the engineer to efficiently drill down to the problem's cause.

Two types/purposes for trace software events

Two completely separate uses for trace software events are common. These are:

1. Emit data from a program not available from a PTM (micro-level detail)
2. Mark major system state changes (macro-level detail)

In some cases, Embedded Trace Macrocells (ETM) may serve most of the micro use cases. However, when instruction tracing is accomplished with a PTM (or an ETM with no data, such as on a Mx), STM is very useful for simply tracing select program data values.

System state changes (with values included) are often logged via a software mechanism. STM is more efficient and it correlates on a time scale to the PTM or ETM instruction trace streams. Examples of interesting and common macro events are:

1. Power state changes
2. ECC failures
3. Buffer overflow errors
4. Entry into a major functional area of a program

5. Semaphore/mutex acquisition/release

These types of events label the overall state of the system and allow the engineer to follow the state transactions without looking at the code minutia.

STM system architecture

Below is a block diagram (Figure 3) of a dual-core SoC with ARM A9 and Cortex R4 cores, as well as a digital signal processor (DSP) core. In this architecture, any of the A9s, R4 or DSP cores can write to the STM. In addition, STM also monitors several hardware signals.

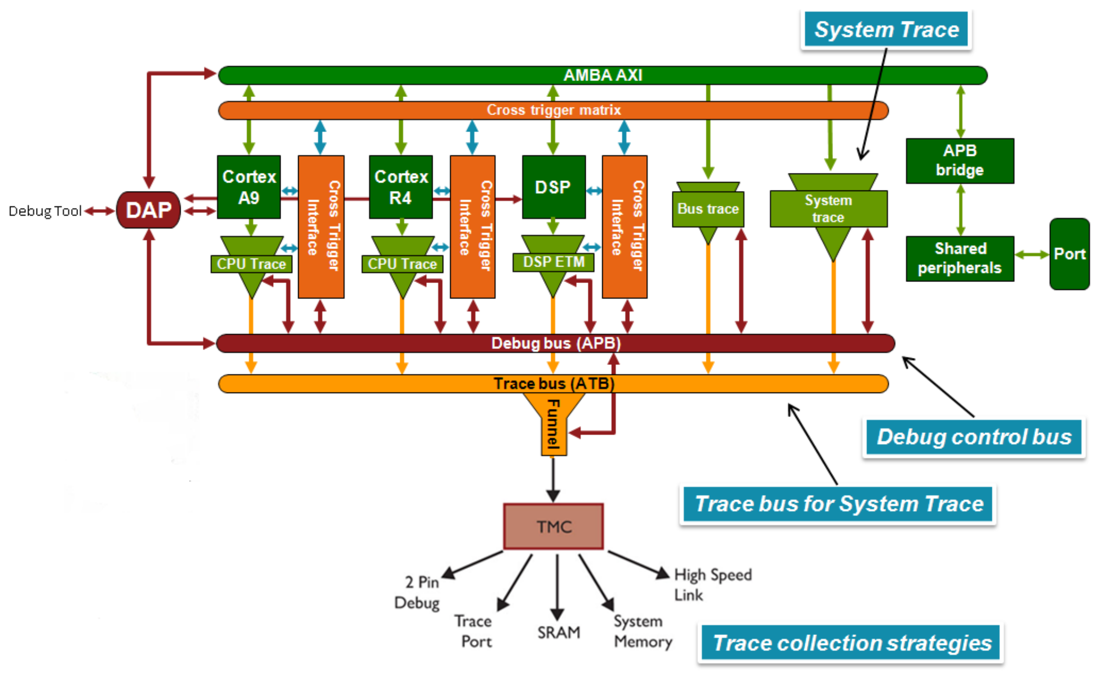


Figure 3: Block diagram of SoC with STM

STM: A new programming paradigm

Before an STM can produce trace messages that will clearly track system states, software engineer(s) must add instrumentation code to their programs. This STM-specific code may only be included in debug builds of the product's software or it may be present in all forms of the system's software. Usually, some STM code will be present in the software. Because the STM is a very efficient way to instrument trace in software, more of the STM instrumentation code can be left in the production software than was previously practical with older trace methods. This is possible because the STM provides no backpressure to the code, meaning the code will continue to execute regardless of whether a trace-sink or adequate trace-sink bandwidth are available. When a trace-sink or the necessary trace-sink bandwidth are not available, trace cycles may be lost but the program does not stall. This is very different from previous trace methods, which directed printfs statements to serial channels.

Several STM use case models will evolve, depending both on the history of the products being debugged as well as product features.

STM use case models

Depending on the features of the product as well as the history of the product's code base, several possible use case models will be followed for instrumenting code for STM tracing.

The code base history issue is the result of legacy logging methods present in large code bases. For example, it will be very unlikely that anyone would recode all of the kprintfs in a particular version of the Linux kernel. That said, capturing these output streams in an STM-based trace transport system still has huge advantage. These implementations, while being much less efficient than new code based exclusively on STM-optimized use models, will still offer much more time efficiency in terms of processing cycles and performance than not including STM trace in the code at all. The opposite alternative would be to develop totally new code with STM tracing and not be saddled by the inefficiencies of the legacy printf function.

On a completely different axis, the way that the code runs produces very different requirements. For example, symmetrical multi-threading code would have different requirements from single-threaded fixed location code. For these reasons, one should consider the six different programming models shown in Table 1.

Table 1: Programming Models

	New Code (Little or no trace messaging present)	Legacy Code (Large numbers of trace messages (printfs) present)
OS (Linux) Process	1b. Kernel call to access master/port; master may be irrelevant (SMP); no printfs	1b. Lots of printfs
Flat OS (including Linux kernel only)	2b. Static code but SMP and dynamic master and port assignments may be required (processes running)	2b. Lots of kprintfs
Bare metal	1b. Fully static; compile time assignments for master and port are allowed.	3b. Lots of printfs (or possibly ITM writes)

Tool-hosted printf capabilities

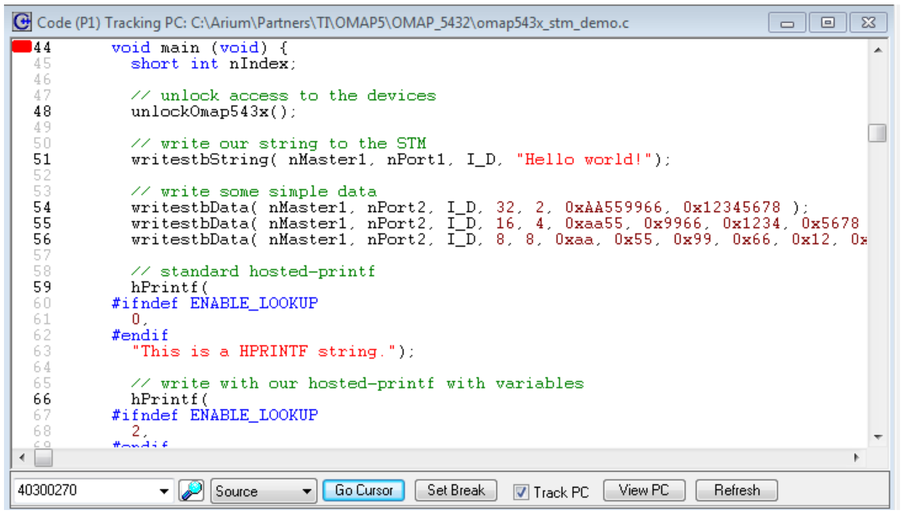
Today, advanced debugging tools are able to make logging through an STM even more efficient. This feature is often referred to as tool-hosted printf. It moves almost all the overhead of printf to the debug tool and outside of the user's product. The only things that pass through the STM are parameters (variables) and a pointer to a printf style format string. This offers two major advantages:

1. Smaller payload through the STM (fewer write cycles)
2. Less processing to prepare a string (faster target execution time)

This method is applicable to all of the use cases outlined in the previous section. This feature should be considered for all new code bases where it is possible to use an STM.

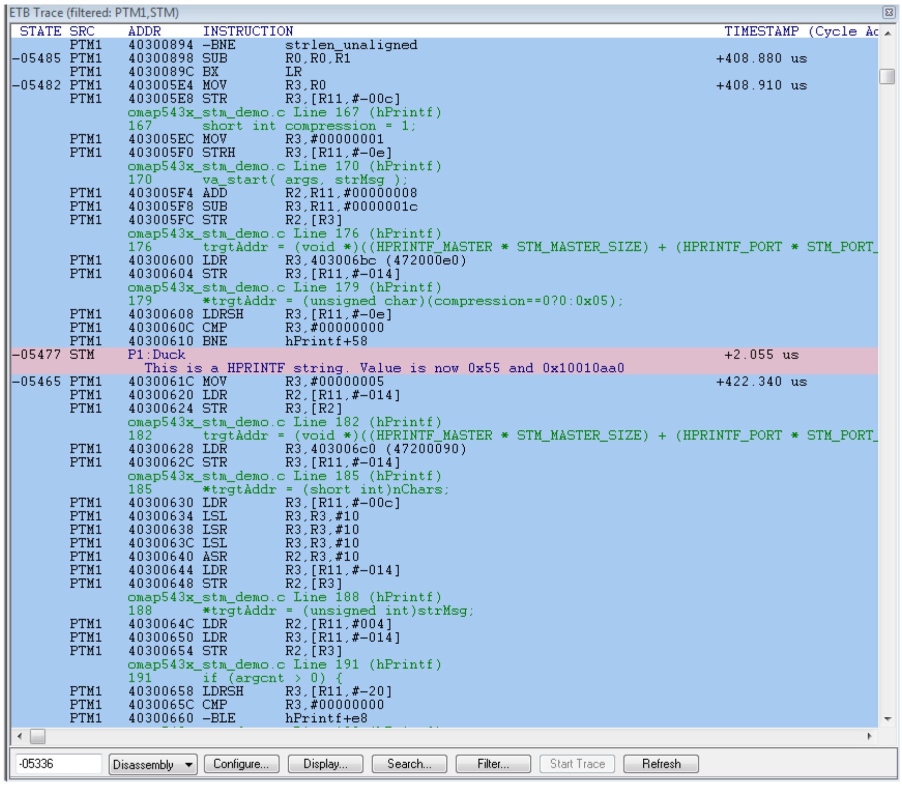
- Examples of tool-hosted printf

Arium™ SourcePoint™ debuggers from ASSET InterTech are the only tools that currently support tool-hosted printf capabilities (patent pending). The following images show some very simple code that is writing data to an STM and then displaying some of that data along with PTM instruction trace data. Figure 4 shows some code with both simple writes directly to the STM as well as tool-hosted printf output. Figure 5 shows a trace window, tracing a portion of this code including the tool-hosted printf output. Notice that the output is delayed until the entire message has been transmitted, but it is still time stamped so that the beginning write can be located.


Code (P1) Tracking PC: C:\Arium\Partners\TI\OMAP5\OMAP_5432\omap543x_stm_demo.c
40300270
Source
Go Cursor
Set Break
Track PC
View PC
Refresh

```
	void main (void) {
	  short int nIndex;

	  // unlock access to the devices
	  unlockOmap543x();

	  // write our string to the STM
	  writestbString( nMaster1, nPort1, I_D, "Hello world!");

	  // write some simple data
	  writestbData( nMaster1, nPort2, I_D, 32, 2, 0xAA559966, 0x12345678 );
	  writestbData( nMaster1, nPort2, I_D, 16, 4, 0xaa55, 0x9966, 0x1234, 0x5678
	  writestbData( nMaster1, nPort2, I_D, 8, 8, 0xaa, 0x55, 0x99, 0x66, 0x12, 0x

	  // standard hosted-printf
	  hPrintf(
	#ifndef ENABLE_LOOKUP
	  0,
	#endif
	  "This is a HPRINTF string.");

	  // write with our hosted-printf with variables
	  hPrintf(
	#ifndef ENABLE_LOOKUP
	  2,
```

Figure 4: Writes to STM and tool-hosted printf output



```
 STATE SRC    ADDR     INSTRUCTION                                                           TIMESTAMP (Cycle Ac
       PTM1   40300894 -BNE       strlen_unaligned
-05485 PTM1   40300898 SUB        R0,R0,R1                                                  +408.880 us
       PTM1   4030089C BX         LR
-05482 PTM1   403005E4 MOV        R3,R0                                                     +408.910 us
       PTM1   403005E8 STR        R3,[R11,#-00c]
              omap543x_stm_demo.c Line 167 (hPrintf)
              167      short int compression = 1;
       PTM1   403005EC MOV        R3,#00000001
       PTM1   403005F0 STRH       R3,[R11,#-0e]
              omap543x_stm_demo.c Line 170 (hPrintf)
              170      va_start( args, strMsg );
       PTM1   403005F4 ADD        R2,R11,#00000008
       PTM1   403005F8 SUB        R3,R11,#0000001c
       PTM1   403005FC STR        R2,[R3]
              omap543x_stm_demo.c Line 176 (hPrintf)
              176      trgtAddr = (void *)((HPRINTF_MASTER * STM_MASTER_SIZE) + (HPRINTF_PORT * STM_PORT_
       PTM1   40300600 LDR        R3,403006bc (472000e0)
       PTM1   40300604 STR        R3,[R11,#-014]
              omap543x_stm_demo.c Line 179 (hPrintf)
              179      *trgtAddr = (unsigned char)(compression==0?0:0x05);
       PTM1   40300608 LDRSH      R3,[R11,#-0e]
       PTM1   4030060C CMP        R3,#00000000
       PTM1   40300610 BNE        hPrintf+58
-05477 STM    P1:Duck                                                                        +2.055 us
                This is a HPRINTF string. Value is now 0x55 and 0x10010aa0
-05465 PTM1   4030061C MOV        R3,#00000005                                              +422.340 us
       PTM1   40300620 LDR        R2,[R11,#-014]
       PTM1   40300624 STR        R3,[R2]
              omap543x_stm_demo.c Line 182 (hPrintf)
              182      trgtAddr = (void *)((HPRINTF_MASTER * STM_MASTER_SIZE) + (HPRINTF_PORT * STM_PORT_
       PTM1   40300628 LDR        R3,403006c0 (47200090)
       PTM1   4030062C STR        R3,[R11,#-014]
              omap543x_stm_demo.c Line 185 (hPrintf)
              185      *trgtAddr = (short int)nChars;
       PTM1   40300630 LDR        R3,[R11,#-00c]
       PTM1   40300634 LSL        R3,R3,#10
       PTM1   40300638 LSR        R3,R3,#10
       PTM1   4030063C LSL        R3,R3,#10
       PTM1   40300640 ASR        R2,R3,#10
       PTM1   40300644 LDR        R3,[R11,#-014]
       PTM1   40300648 STR        R2,[R3]
              omap543x_stm_demo.c Line 188 (hPrintf)
              188      *trgtAddr = (unsigned int)strMsg;
       PTM1   4030064C LDR        R2,[R11,#004]
       PTM1   40300650 LDR        R3,[R11,#-014]
       PTM1   40300654 STR        R2,[R3]
              omap543x_stm_demo.c Line 191 (hPrintf)
              191      if (argcnt > 0) {
       PTM1   40300658 LDRSH      R3,[R11,#-20]
       PTM1   4030065C CMP        R3,#00000000
       PTM1   40300660 -BLE       hPrintf+e8
```

Figure 5: Trace window with tool-hosted printf output

Ease-of-Use

XML-based metadata files will allow a given project to customize the STM display lines such that the labeling and ASCII notes make the activity clear. This must be on a master/port basis. This should make both the size of the program being debugged and the programmer’s effort as minimal as possible. For simple tabular displays or simple strings, this can all be done in the metadata. For more varied information, tool-hosted printf capabilities can maximize efficiencies. The following is the metadata that was used in the previous examples. Clearly, much of the data labeling can be done on a per port basis, requiring no additional text via the STM.

Example of XML-based metadata:

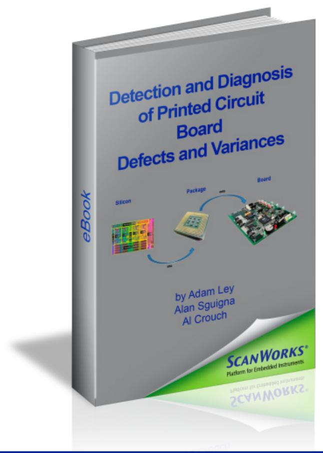
```
<?xml version="1.0" encoding="utf-8"?>
<!--
   File:    ports.xml
   Purpose: SourcePoint sample STM metadata
-->
<settings>
    <setting name="mark terminates message" value="false"/>
    <setting name="newline terminates message" value="false"/>
</settings>
<masters>
    <master id="0x40" name="P0" format="ASCII">
          <ports>
              <port id="0x43" name="Blue"/>
              <port id="0x25" name="Norwegian" format="Tabular Hex"
columns="16"/>
                    <port id="0x2000" name="Parrot" format="Hosted
printf"/>
          </ports>
    </master>
    <master id="0x41" name="P1" format="ASCII">
          <ports>
              <port id="0x43" name="Green"/>
              <port id="0x25" name="Swedish" format="Tabular Hex"
columns="16"/>
              <port id="0x100" name="Timer_Placeholder" format="ASCII"/>
                    <port id="0x2000" name="Duck" format="Hosted
printf"/>
          </ports>
    </master>
</masters>
<hardware>
    <event id="0" name="signal1"/>
    <event id="1" name="signal2"/>
    <event id="8-15" name="bus"/>
```

Conclusions

The capabilities of STM such as event logging in software through the STM can be a real game changer for software debug in complex programmed systems. Looking at a filtered version of an STM trace stream will show the engineer only a high-level abstraction of the system state. Once the area of trace interest is located, the engineer can then drill down to whatever level of detail he needs to find the root-cause of a system fault. In addition to STM's software-generated events, it also supports hardware stimulus ports that can monitor specific signal changes within the SoC. Modern SoC design should not be considered today without at least a thorough examination of the usefulness of including one or more STMs in the SoC. STM, with its post-processing tools, can often save weeks in a product development cycle. This can produce significant cost savings, both direct and indirect.

Learn More

If you are interested in "Board Test of DDR3/DDR4 Memory and Serial I/O", you should check out this tutorial.


Detection and Diagnosis of Printed Circuit Board Defects and Variances
eBook
by Adam Ley
Alan Sguigna
Al Crouch
ScanWorks

Register Today!

Glossary:

APB – Advanced Peripheral Bus

ATB – Advanced Trace Bus

AXI - Advanced eXtensible Interface

CoreSight – An ARM® architecture for debug components on and SoC

CTI – Cross Trigger Interface

DSP – Digital Signal Processor

ECC – Error Correction Code

ETM – Embedded Trace Macrocell

MIPI – Mobile Industry Processor Interface as defined by the MIPI Alliance

PTM – Program Trace Macrocell

SOC – System-on-a-chip

STM – System Trace Macrocell

STP – System Trace Protocol

XML – eXtensible Mark-up Language